Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Jonas Bakken

Parameter-based online optimization in metric indexing using Bentley-Saxe transformations

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
June 2021

NTNU
Norwegian University of
Science and Technology

Jonas Bakken

Parameter-based online optimization in metric indexing using Bentley-Saxe transformations

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Abstract

Metric indexing is an important tool when dealing with similarity searches of large datasets. Many of the algorithms and data structures in this area rely on some form of a parameter in the indexing process. The optimal choice for this parameter is often determined by the dataset that is being indexed, and choosing some non-optimal parameter will in some cases have a large impact on the performance of the index. This report introduces the BS-OPTPARAM, a general approach to finding the optimal parameter for dynamic metric indexing data structures that leverages Bentley-Saxe transformations. The paper also describes and tests four different algorithms that use the BS-OPTPARAM approach. Some promising results are presented, showing some cases where the algorithms perform better than expected, and indicating that the BS-OPTPARAM is a useful tool when solving this problem.

Sammendrag

Metrisk indeksering er et viktig verktøy for å kunne søke i store datamengder. Mange algoritmer og datastrukturer i dette feltet bruker en form for parameter i indekseringsprossessen. Det optimale valget for denne parameteren er i mange tilfeller avhengig av datasettet som blir indeksert. Denne rapporten introduserer BS-OPTPARAM, en generell metode for å finne den optimale parameteren for dynamiske datastrukturer brukt i metrisk indeksering. Rapporten beskriver og tester fire forskjellige algoritmer som bruker BS-OPTPARAM metoden. Det blir presentert noen lovende resultater hvor algoritmene presterer bedre enn forventet, som tyder på at BS-OPTPARAM er en nyttig metode for å løse problemet.

Contents

Chapter 1

Introduction

Similarity search and metric indexing is a useful and often necessary tool when handling large amounts of data, and has been successfully applied in numerous fields. There exists a large number of algorithms and data structures that solve this problem, all with its own set of advantages and disadvantages. Some are relatively straightforward and work sufficiently well for most applications, while others allow for fine-tuning of different parameters to get the best performance possible in a given scenario.

Some of these data structures are dynamic, allowing efficient insertions and deletions of elements, while a lot of them are static, requiring the entire dataset that is going to be used when building the index. While the dynamic data structures can be used in all the same applications as the static, it is not true the other way around. Allowing efficient insertions makes the data structures suitable for a whole other set of applications. The static data structures will on the other hand often give better performance, as the index is built with full knowledge of the data it contains. Because of this, it is interesting to look at transforming these static data structures to dynamic, allowing for a compromise between the two.

This project looks at taking advantage of how a specific static-to-dynamic transformation works in order to optimize the metric index by fine-tuning the parameters of the index to the elements it contains. The general approach presented is called BS-OPTPARAM, and four algorithms using this approach are presented, implemented, and tested.

In the fall of 2020, I did some preparatory work to this project where I tested the feasibility of the idea, and tried to discover some of the challenges that would come up when creating these algorithms. That project will later be referred to as the preliminary study to this report. Some material is taken directly from the preliminary, namely figures 2.1, 2.2 and 5.1, as well as some of the simpler implementations of metric indexing data structures and the Bentley-Saxe transformation, which were suitable to be reused here.

I consider the contributions of this project to be:

- The groundwork presented in chapters 2, 3 and 4.
- The idea behind the general BS-OPTPARAM approach, presented in chapter 5.
- The four different BS-OPTPARAM algorithms and implementations presented in chapter 6:
  1. Canary
  2. Selection
  3. RARR
  4. SSA
- The results gathered from the experiments and the related discussion, in chapters 7 and 8.

Chapter 2

Background

2.1 Metric Indexing

A similarity search is, as the name suggests, a search where one wants to find objects similar to some specific query object. The interesting properties of the elements are therefore the relation between elements, not some inherent property of the element itself. This is opposed to other search problems, where the objective may be to retrieve a specific element from a set.

Metric indexing looks at similarity searches where the relation between the elements are defined by a *metric distance function*, or simply metric[1]. The metric $d : U \times U \to \mathbb{R}_{\geq 0}$ takes two elements and returns the distance between them. In addition, the metric function needs to satisfy the following properties:

$$\begin{gathered} d(u,v) = 0 \Longleftrightarrow u = v \\ d(u,v) = d(v,u) \\ d(u,v) \leq d(v,w) + d(w,u) \end{gathered}$$

The Euclidean distance function is an example of a metric that describes the distance between two points in n-dimensions. These metrics can also be created for applications that don't have such an intuitive way of assigning a distance, such as the Levenshtein distance, or string edit distance, that describes the minimum amount of "edits" are required to turn one string into the other.

The simplest query in a metric index is the *ball query*[2], consisting of a center $q \in U$, and a radius $r \in \mathbb{R}$. The result of the query contains all elements with distance to q less than r; $Q = \{u : d(q,u) \leq r\}$.

The metric index itself is a preprocessing of these elements so that future queries are faster. A common way to achieve this is to divide the elements into regions based

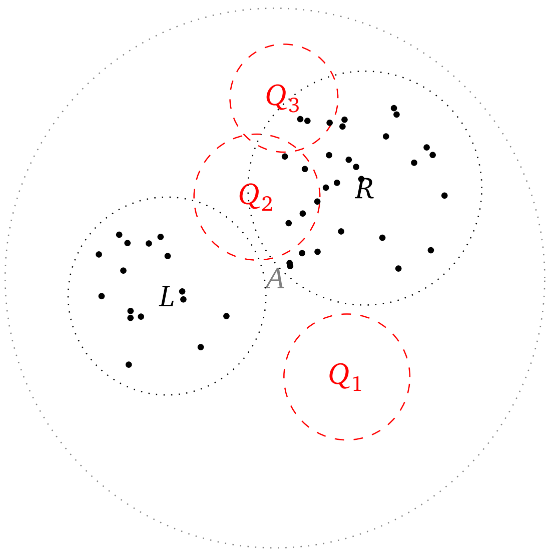

Figure 2.1: Three cases when performing a query

on their distance to each other, and when querying the index, some regions may be ignored if they don't overlap with the query. This is illustrated in figure 2.1, where the area A is divided into two child regions L and R. We see that the query Q_1 is outside both regions, and we can say that the query returns no elements. Q_2 overlaps with both L and R, so both regions need to be searched. Finally Q_3 only overlaps with R, so we can ignore L and only continue searching the one region.

2.1.1 Vantage point tree and hierarchy of clusters

The vantage point tree[3] (VP-tree) is a binary tree. It is built by first selecting one *pivot* element p, the "vantage point," and then dividing the rest of the elements into two regions, the left and right child. The distance from p to each element is evaluated to find the median m. The elements are divided so that the half with distance less than m is assigned to the left child, and the rest to the right. The procedure is then called recursively for the left and right child, making a balanced binary tree.

The hierarchy of clusters[1] (HC) is built similarly to the VP-tree, but instead of always assigning the closest half of the elements to the left child, the number of elements are decided by a function $h(n) : \mathbb{N} \to \mathbb{N}$. Fredriksson suggests using $h(n) = n^{\frac{1}{\alpha}}/2$ or $h(n) = \frac{n}{\alpha}$. This parameter α decides how imbalanced the tree should be. Note that using $\alpha = 1$, or $\alpha = 2$ for the second function, yields the same tree as when using the VP-tree.

From the normal binary tree search, we may expect a balanced tree to perform better in general. This is however not always the case with metric indexing, and experimental results show that an imbalanced HC will often outperform the VP-tree[1].

2.1.2 Sparse spatial selection tree

The sparse spatial selection tree[4] (SSS-tree) is a tree where each pivot has a variable number of children or fanout. The tree is built by first selecting an element p randomly and adding it to a list of *cluster centers*. For each element, the closest cluster center is found. If the distance is less than αM, it is assigned to that cluster, otherwise, it is added as a new cluster center. α is a user-defined parameter, while M is the maximum distance between any two elements in the set. Brisaboa *et al.* suggests that the optimal choice for α usually is found in the range $[0.35 - 0.4]$. The process is repeated recursively for each cluster.

2.2 Static-to-dynamic transformation

Some data structures are dynamic, allowing efficient insertions and deletions. Others, like the binary search tree, allows for insertions and deletions, but the performance may degrade as a result. And some data structures, like the SSS-tree or VP-tree/HC, don't allow insertions without rebuilding the entire data structure.

The topic of static-to-dynamic transformations is interesting as it allows for static data structures to be used in new applications. Static data structures are also interesting in themselves as they are built with complete knowledge about the data it contains, which may for example allow for better guarantees about complexity. Experimental results show that even with the added overhead of the transformation, static data structures used as dynamic can perform on par or even better than their dynamic counterparts[5].

2.2.1 Bentley-Saxe transformations

Bentley and Saxe proposed a set of transformations that allows any static data structure solving *decomposable problems* to be used dynamically. A decomposable problem is a problem where for any two datasets $\mathbb{D}_1$ and $\mathbb{D} \setminus \mathbb{D}_1$, the result of an operation (read query) over $\mathbb{D}$ can be achieved by a combination of the same operation over $\mathbb{D}_1$ and $\mathbb{D} \setminus \mathbb{D}_1$[7].

All the transformations rely on keeping track of multiple smaller data structures, referred to as buckets, and rebuilding them in a pattern that yields a low amortized complexity both for insertions and queries[6]. The way the transformations differ is in how this pattern of rebuilds is done. All are based on different representations of numbers, with the simplest being the binary transform.

The binary transform rebuilds buckets in a pattern similar to how counting in binary representation works, as illustrated by figure 2.2. On each insertion, the elements

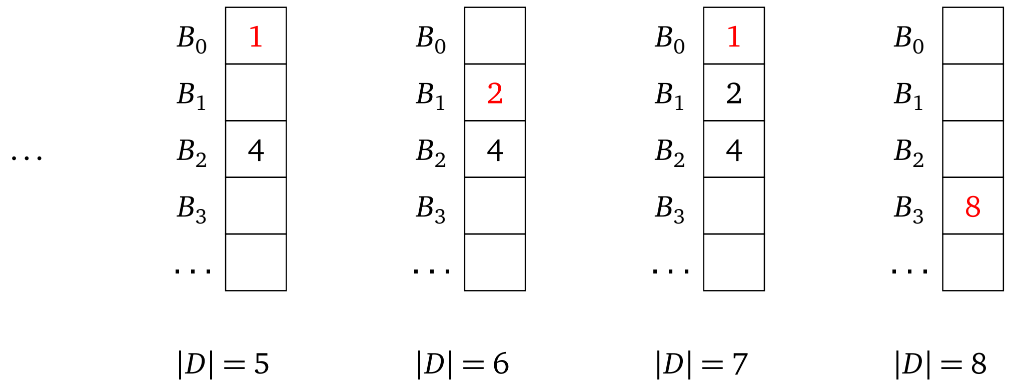

Figure 2.2: Binary transformation on insertion 5 to 8

of all the smallest buckets together with the newly inserted element are combined into a new, larger data structure and placed into the next bucket. While some insertions are expensive, namely the ones where the entire dataset has to be rebuilt, the amortized cost is kept low by the fact that most insertions only end up rebuilding a relatively small number of elements. When querying, there is an added cost of having to query each bucket individually, but this is also kept relatively small by the fact that there will be at most $\log_2 n$ buckets to query, most of them smaller.

Another interesting transformation is the k-binomial transform, which keeps the number of buckets constant which results in cheaper queries, at the cost of a more expensive insertion.

Chapter 3

Problem and motivation

The idea behind this project stems from two observations that came when trying to implement and benchmark a dynamic version of the SSS-tree using the binary Bentley-Saxe transformation.

The first observation is related to the implementation of an SSS-tree. When building an SSS-tree, the parameter α decides how each region should be divided into child regions, and its optimal value is dependant on some property of the dataset used, namely the intrinsic dimensionality[8]. This means that an easy way to decide what the α should be is to test out different choices for α on a small subset of the dataset and choose the best one. But how should this be done when using a dynamic version of the SSS-tree, when you don't know what the dataset will look like? The trivial answer is to use a dataset that is similar to the one that will be used when inserting items into the tree and testing the same way one would do with the static data structure. But the dataset you choose may not match the one that will be inserted into the data structure, because if all the points were known ahead of time, there would not be a need for a dynamic data structure. There is an inherent difference in the set of elements contained in a dynamic data structure at any point in time, and the set of elements you can test for beforehand.

If you know that you will be indexing some set of English words, you could select a parameter based on tests of the English dictionary. This is a case where you know the largest possible superset of the elements that will be inserted, but there is no guarantee that the elements inserted will match the dimensionality of the entire dictionary. This fact can also be demonstrated by the case where you know that all elements that will be inserted lie within the three-dimensional unit cube. If we imagine that an adversary chooses what elements are inserted, the adversary can for example choose elements on a line, say $y = 0$, $z = 0$, and the dimensionality of the elements in the data set differs from the one we optimized for. So the problem of finding a suitable α for a dynamic data structure still remains unsolved.

The second observation is related to the application of Bentley-Saxe transformations with the SSS-tree. It is based on the fact that when using a Bentley-Saxe transformation, there is often a large number of smaller buckets that make up a small portion of the total number of elements. These small buckets account for a small portion of the total complexity of the data structure, but they are rebuilt multiple times for each time a larger bucket is built. The idea is that one could do "experiments" on the smaller buckets to find out what the parameter should be for the larger buckets that make up a more significant part of the complexity. These tests could be done on elements that are inserted into the tree, and may therefore give a better insight into the properties of the entire set used. And because the smaller buckets make up such a small portion of the total number of elements, these experiments may be done without a large footprint.

Hypothesis 1 *For many applications (like indexing) using a Bentley-Saxe transformation, choosing some non-optimal parameter α for some number of smaller buckets $B_0, B_1, \ldots, B_n$ with $n < m$ where m is the total number of buckets, will have little effect on the overall efficiency, both queries and insertions, of the data structure.*

The use of the Bentley-Saxe transformations allows for both collecting information about the currently used dataset and using this information to change what parameter is used. In many cases, the only way to change what parameter is used in a data structure is to rebuild the entire data structure with the new parameter. This is a costly operation when applied to the entire dataset, but the Bentley-Saxe transformation relies on complete rebuilds and performs them in a specific pattern to keep the complexity manageable.

All in all, the problem that is investigated in this project is at the intersection of:

- Online optimization
- Static-to-dynamic transformations
- Metric indexing

Chapter 4

Related work

Although there has been done extensive research in the area of metric indexing, it is difficult to find literature related to automatically tuning the algorithm's parameters. Even though multiple metric indexing methods use some form of parameters that affects the performance, the process of setting these parameters is often manual and based on trial and error[9].

Fredriksson describes ways of improving different metric indexes. This is where the HC came from, as an improvement over the List of Clusters[10]. The paper also provides other interesting algorithms that improve upon different existing data structures such as the Burkhard-Keller tree[11]. These algorithms do however only focus on static data structures. One thing that is worth pointing out is how the paper discusses handling unknown query sizes, as the optimal parameter for the HC is sensitive to the range of the query. The paper suggests building multiple instances of the data structure, tailored to different radii, and shows experimentally that only a few instances are needed[1].

In the paper about the VP-tree, Yianilos describes a way of optimizing it for a given data set. This is based on the insight that each element has its own so-called perspective of the rest of the space. This perspective is formed by looking at the distances from the object, to all others. It is pointed out that even though all elements have a perspective, they do not necessarily contain much, or possibly any, information. This is for example the case when the distances to all other objects are the same, and an exhaustive search is unavoidable. The paper considers, among other things, using this insight to make better pivot selections. This work differs from what is discussed in this paper, but there may be merit in exploring using this type of optimization in combination with the Bentley-Saxe transformations.

Muja and Lowe looks at, among other things, how some algorithms perform well on some data sets, but poorly for others. The paper views different approximate nearest neighbor algorithms as different parameters to a generic search routine and optimizes the parameter selection for a given data set. This optimization is done statically, for a known data set. This work differs in that the paper describes a method of finding the optimal choice for a given data set, while the BS-OPTPARAM tries to optimize for an unknown data set.

Dong *et al.* looks at Locality-Sensitive Hashing[12] and provides a method for auto-tuning parameters of the search algorithm, in order to achieve better results for this approximate similarity search algorithm. Jääsaari *et al.* also looks at improving auto-tuning of different approximate nearest neighbor search algorithms but focuses on a handful of randomized space-partitioning trees. Even though both papers look at auto-tuning parameters for approximate similarity search, the BS-OPTPARAM differs in that it is a more general method, not a variation of an existing data structure.

Chapter 5

Solution outlined

The idea, as previously stated, is based on the idea that smaller buckets are used to test out different (possibly less optimal) parameters, in an attempt to gather information about how the different parameters respond to the dataset. This information can then be used to make informed guesses about what could be a more optimal choice, and apply it to larger buckets. This general method to solve the problem will be referred to as *BS-OPTPARAM* and can be approached in several ways. Some specific BS-OPTPARAM strategies and algorithms will be discussed in the next chapter, but some of the groundwork will be laid here.

In order to explore the merits of the BS-OPTPARAM, the following research questions were proposed:

RQ 1 *How can the Bentley-Saxe transformation be usefully applied in the optimization of an underlying data structure?*

RQ 2 *When is there a significant enough difference between the performance of different parameters for this type of optimization to be worthwhile?*

The first research question is meant to explore how one should go about using the BS-OPTPARAM approach to create an algorithm. The second question is meant to explore when this approach is useful.

5.1 Assumptions

Although one can imagine that the BS-OPTPARAM could be applied to a number of different use cases, also outside of metric indexing, this report works with strategies that are designed with the following assumptions in mind for the underlying data structure and problem.

- Decomposable problems.
- The build operation is asymptotically more expensive than the query operation.
- The parameter can't be changed without rebuilding the data structure.
- The data structure uses a parameter where its performance is based on the data contained in the data structure.
- The potential gain from selecting a "good" parameter can outweigh the extra overhead related to these "experiments".
- There is some way to relate an optimal parameter for a small set to a larger one when taken from the same dataset.
- The dataset is not known.

Looking at the assumptions, the first three could be considered normal for most uses of the Bentley-Saxe transformations. The fourth and fifth sets constraints to which data structures the approach is applicable. These assumptions are considered to be fair, as the approach, in general, is based on achieving better results by simply selecting a better parameter. The sixth assumption is based on the fact that if the premise of these strategies is correct, that a somewhat optimal parameter for larger buckets can be found by testing smaller buckets, then there needs to be a way to relate observations from the small bucket to predictions about a larger bucket. The final assumption could be considered unrealistic, as one will probably be able to get some insight into the datasets used in real-world use cases. The assumption is based on the fact that none of the strategies should rely on knowledge about what parameters work well, or don't, for any specific dataset.

Based on these assumptions, more specifically the fourth and fifth, the HC is a better choice for underlying data structure than the SSS-tree when used in the BS-OPTPARAM. This is because the span of the optimal parameter for the SSS-tree is small, and the difference achieved within that range is minimal[8]. The HC on the other hand has a large span of useful parameters, and the cost of selecting a non-optimal parameter can be significant[1].

5.2 Specification

To be able to more accurately describe the different approaches to the BS-OPTPARAM and expected challenges, some terminology will be introduced.

5.2.1 Picking method

When using an indexing structure in an online problem, I want to be able to describe how items are added to the index. From a universe U, items are picked and added to the index following a *picking method*. The picking method describes how an item is selected in relation to the previously selected items. Selecting a random element

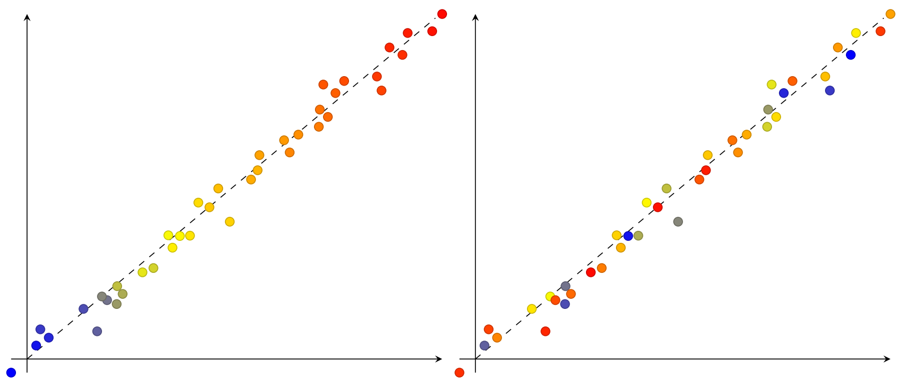
Figure 5.1: Local picking method (left) vs uniform (right) for the same set of points

for insertion, unrelated to the last, can be described as a *uniform/random picking method*. Selecting the next element based on a relation to the previous element, for example proximity, can be described as a *local picking method*.

Using a uniform picking method will intuitively create subsets of U that *look like* U. One would think that they preserve the spacial properties of U, like the intrinsic dimensionality. This intuitively applies to all randomly selected, sufficiently large, subsets. This is an appealing property, as we would assume that small and large buckets resulting from this picking method, would yield the same optimal parameter.

Hypothesis 2 *Uniform picking methods yield approximately the same optimal parameters for any sufficiently large bucket.*

For a local picking method, however, this may not be true. Figure 5.1 shows the same set of points using two different picking methods, where blue elements are inserted early, and red are selected later. Considering the two picking methods, it seems like it would make sense that the nature of the universe U is not as clearly reflected in the local picking method compared to the uniform when only considering a small subset of the points. This would imply that differently sized subsets can have different properties and that the resulting optimal parameter may therefore be different at different scales.

Hypothesis 3 *Local picking methods can yield different optimal parameters as new elements are selected.*

5.2.2 Bentley-Saxe

When discussing a Bentley-Saxe transformation, the binary transform should be assumed unless otherwise stated. A bucket B_i is the i^{th} bucket, with $i = 0$ being the first/smallest. A bucket is either *full*, containing 2^i elements, or *empty*, containing 0.

The *initial build* is the first time a bucket is filled with elements, any time a bucket is filled with elements is a *build* or *rebuild*. All buckets that give their elements to B_i in a build are the predecessors of B_i, and B_i is the successor to those buckets. Note that for a given bucket, the predecessors will always be the same (all the smaller buckets), while the successor will change depending on the build. The entire data structure is $\mathcal{B}_n = [B_0, \dots, B_n]$. The number of elements contained there are:

$$|\mathcal{B}_n| = \sum_{i=0}^{n} |B_i| \leq \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

This is an equality when all buckets are full.

5.2.3 BS-OPTPARAM

Potential parameters are parameters that are optimal for some input, as opposed to parameters that never perform well ($\alpha = 0$). These are the parameters worth considering, given no knowledge about what data can be inserted.

An *offline* or *static* solution is one that selects a parameter before knowing any elements, and sticks to it. A *semi-static* solution can look at some elements to get some insight before choosing a parameter, but then it is set. An *online/dynamic* solution can change parameters at any time.

To be able to describe, benchmark, and compare BS-OPTPARAM implementations across strategies, a *strategy cost* is used. This describes the additional cost introduced by implementing a specific BS-OPTPARAM strategy/algorithm, compared to just using the normal Bentley-Saxe algorithm as a baseline. This cost is negative if implementing the strategy yields a better result than not doing so.

There are multiple options for what the baseline should be, in terms of what choice of parameter the baseline should use. The choice can be dependant on what the alternative to the BS-OPTPARAM is, and some logical choices are:

- Comparing to offline solutions by calculating cost compared to what could have been achieved when selecting a (possibly random) potential parameter. This can be done by comparing against the average of a set of potential parameters that the dynamic solution also can choose from.
- Comparing to semi-static solution by running queries on a small number of test elements, and choose the best performing parameter as the baseline. A simple way to get an estimation of this is to compare against the best static choice of parameter.
- The last option is to compare the cost to that which could have been achieved by some omnipotent/oracle algorithm that always rebuilds buckets with the optimal parameter. This cost will always be positive and is only useful when comparing online strategies to each other, or when looking at the potential of a given implementation.

The first and second options will be used in this report, both comparing to the average and best static choice. The average is included because if there truly is no knowledge about the dataset, the only thing to do is guess a potential parameter. The best static choice is included because that is what is usually used for comparisons in the analysis of online problems when using multiplicative weights, which has similarities to the idea of dynamic solutions. This also gives us a better insight into what can be achieved in a more realistic scenario where we actually can make some assumptions about the data set.

These two types of strategy costs can then be described as:

$$C_{\text{Average}} = \text{achieved cost} - \text{average static cost}$$

$$C_{\text{Best}} = \text{achieved cost} - \text{best static cost}$$

When larger amounts of data are used, the magnitude of the cost increases. So to be able to compare costs across different bucket sizes, the *relative strategy cost* is used.

$$R_{\text{Average}} = \frac{C_{\text{Average}}}{\text{average static cost}}$$

$$R_{\text{Best}} = \frac{C_{\text{Best}}}{\text{best static cost}}$$

The *explicit cost* of a BS-OPTPARAM strategy is the overhead added by the strategy to do other operations than what is needed when using a simple Bentley-Saxe transformation. The *implicit cost* is the cost added by selecting a non-optimal parameter.

A *top(bottom)-heavy* data structure is one where most of the full buckets are big(small). The data structure is *sparse* when few buckets are full.

A *query-time* strategy associates the experiments and observations with the query operation of the data structure, and the extra overhead of the solution is therefore added to queries. An *insert-time* strategy associates it with the insert operation, and therefore adds overhead to insertions.

5.3 Challenges

As discussed in the preliminary study and earlier in this report, I expect there to be a number of challenges associated with trying to implement a solution to this problem:

Challenge 1 *Can data be gathered fast enough to be able to obtain a good parameter in a reasonable amount of time?*

Are there enough rebuilds or queries of smaller buckets to get enough data about the performance of a parameter/bucket pair? If the data structure requires an inconveniently large amount of elements or queries or adding significant overhead, just to be able to make an informed guess at what parameters perform well, it limits the usability of the solution. This includes not only being able to gather information about the performance of a specific parameter, but also other potential parameters that may perform better or worse. There is also a significant variance in the performance of different queries on the same metric index. This amplifies the challenge in that enough data needs to be collected to get a representative value for each parameter tested.

Challenge 2 *How does results from smaller buckets relate to the optimal parameter for larger buckets?*

This is an important challenge to handle, as at least half of the total number of elements will be contained in the largest bucket at any time. Therefore if an algorithm uses data collected from smaller buckets to decide the parameter of some successor, the parameter has to be chosen carefully. This large bucket also isn't rebuilt before a new largest bucket already exists, so if the bucket relies on rebuilds to explore different parameters for that bucket, it doesn't have the opportunity to collect data before it is no longer the largest bucket.

This challenge relates to hypothesis 3. While hypothesis 2 expect each bucket to have approximately the same optimal parameter, this this not expected to be true for all cases. This can be exemplified with the left part of figure 5.1. Given an optimal parameter for a data structure containing the blue points, is there an easy way to turn that into a good guess for a data structure containing the yellow, orange, and red points? This certainly seems simpler to do for the uniform picking method.

Challenge 3 *How to efficiently handle changes in the optimal parameter as new elements are inserted?*

This is somewhat related to challenge 2 and 1, as the search space needs to be continually explored in order to be able to observe changes in the optimal parameter for a specific bucket over time. While challenge 2 describes the changes in parameters when the size of the buckets change, this looks at changes within a specific bucket size, when the elements are swapped out with new ones. If we imagine an adversary that first inserts elements forming a low-dimensional set, and then elements forming a high dimensional set, it intuitively follows that the optimal parameter for any bucket will likely need to change.

Challenge 4 *How to collect data about each parameter when the only way to get information about the performance of a parameter is to do a query?*

In this case, we assume that the underlying data structure is a "black box" in terms of performance, in the sense that the only way we can tell how well a certain parameter performs is by executing a query, and getting the result and cost back. This is not always the case. If binary search trees were used, it is possible to look at the balance of the tree to say something about the expected performance, regardless of what elements are searched for. This does however not necessarily apply to metric indexing. While it is possible to draw comparisons to the balance of an HC, and that a more balanced tree (the VP-tree) may perform better in cases with a very low retrieval rate, this is not the same as different queries perform better on different levels of imbalance[1].

Challenge 5 *What to do when the data structure is sparse or top-heavy if strategies rely on testing small buckets for low overhead?*

This challenge is pretty self-explanatory. When using the binary Bentley-Saxe transformation, the data structure will spend about the same time being top-heavy as bottom-heavy. The case where all buckets are full are few and the interval at which they show up increases exponentially over time. The user of the data structure should also not be required to use the data structure in a specific way, meaning that a data structure where just the topmost buckets are full should work.

5.4 Mitigations

This section describes how the different challenges are approached when designing BS-OPTPARAM strategies.

Challenge 1

This is mostly a case of trial and error, and weighing the observed overhead to how well a strategy is going to be at finding the correct parameter. As will be shown later, all proposed strategies can be tweaked in this way. This is a logical first test any strategy has to go through; does there exist compromise between the two where the strategy makes good decisions while the overhead is low enough. This is basically checking if there exists some scenario where the strategy cost is negative, then see if that same tradeoff works for different data sets.

There is still a problem however if the only scenarios found require a number of insertions or queries that could be considered impractical. This imposes rules for the use cases of the data structure and should be kept in mind when examining the results.

Challenge 2

If the optimal parameter is "well-behaved" in regards to the bucket size, there is a hope of finding the relation between small and large buckets. Observations from the preliminary study seem to indicate that this, at least sometimes, is the case. If so, there could be a possibility of for example using regression analysis to find the relation between the different sizes.

Sticky points

If we assume that the reason why translating a good result from a small bucket to a good guess for a large bucket is that the dataset "looks different" on different scales, a possible mitigation is to make the set in each bucket look more like the dataset as a whole. The method of *sticky points* works by "leaving behind" copies of points as they are moved up to successors. Each bucket has a set of $r|B_i|$ "sticky" points, extra points that are added to the data structure when building the bucket, but not transferred to the next bucket on a rebuild and not returned during queries. The effectiveness, and cost, of this method are decided by the *retention factor* r. When a bucket is removed/its points transferred, select $p * r|B_i|$ random (non-sticky) points from it, and add them to the sticky-list, where p is in the range $[0, 1]$. If the list is full, replace random points.

By allowing the list to fill up gradually, we lower the cost of the mitigation, as it takes some rebuilds to fill the list, and by that time several larger buckets will have been built. This means that larger buckets will never have many sticky points, which they shouldn't need as they contain a larger portion of the total dataset.

k-binomal transform

Another, less novel, way this could be handled is by using the k-binomal Bentley-Saxe transformation instead of the binary. There will be the same number of rebuilds, one each insertion, but as the total number of elements will be contained in a fixed number of buckets, the average rebuild will contain a larger portion of the total set. This does however introduce the challenge to the translation from one rebuild to the next in a specific bucket, as each bucket can contain a variable amount of elements.

Challenge 3

To be sure that changes in optimal parameters are observed, the strategy should either continually check multiple parameters where the accepted overhead allows it, or include some probabilistic element to the parameter selection. The cost of random queries varies a lot, and while variance is bad in terms of challenge 1, it could be used as a natural way to introduce some randomness into the algorithm.

Challenge 4

When using metric indexes as the underlying data structure, the natural way of assigning a cost to a query is to run the query and count the number of distance calculations that are done. When and how these queries are executed can however be a difficult choice. For query-time strategies, we can use the queries made by the user to collect data. This allows for a splay-tree[14] type of optimization, where the data structure is optimized according to the way it is being used, possibly at the cost of a more expensive random query. The strategy should however not force a certain use pattern to work, i.e. 100 queries must be done for each new bucket or insertion.

For insertion-time strategies, the data structure still needs to be queried, but the natural point to do it will in this case be in association with an insertion, and therefore the rebuild of a bucket. The assumption that the query operation is less expensive asymptotically than the build operation allows for test-queries without too much of a footprint on the insertion operation. As long as only the bucket that is being rebuilt is queried, and not the entire data structure, the complexity analysis from the Bentley-Saxe transformation can still be used, and the overhead is expected to stay manageable. To create a query, we still need a query center and radius, however. The most natural way to find a center is to just use random elements contained in the bucket. This would allow for something closer to a general balanced tree[15] type of optimization, where the data structure is optimized for random queries. For a choice of query radius, however, we must either just assume a radius, or select one based on earlier queries if there have been any user-provided queries yet. Just assuming the query radius has its own set of implications, but for the sake of the scope of this report, and the fact that other papers do the same[1], I choose to do it.

Challenge 5

All strategies that use results from smaller buckets to make informed guesses about larger ones need some way to handle these cases. A way to do it would be to rely on values accumulated over multiple queries and rebuilds, and therefore not be bound to the data structure being in a specific state for the strategy to work. If accumulating isn't possible, one could just stick to the last decision made with enough data, until there are enough small buckets again.

Another method would be using *retention* of buckets. This would work much like sticky points but differs in that sticky points leave behind points that go on to a successor, while retention leave behind entire buckets.

Chapter 6

Strategies

This chapter proposes four different BS-OPTPARAM strategies. The algorithms are implemented and tested in the next chapter.

When using the HC, the optimal parameter is not only dependant on the dataset, but also the radius of the query. Fredriksson proposes handling this by keeping track of multiple data structures, each with a different parameter. For each query, the appropriate data structure is selected based on the radius of the query[1]. You effectively have a mapping from different radii to the parameters, which is known before the data structure is created and used.

A similar approach is used for the first three strategies described here. We keep track of *p* versions of the same data structure and choose the appropriate parameter for the given input. In this case, however, the parameter should be selected based on the dataset that the index contains, or the dataset in combination with the query, and not the query alone. It is however not that simple to create such a mapping, so the selection will be made based on observations instead. These strategies will naturally come with a *p* factor time increase for insertions, in addition to whatever additional overhead the strategies used to select a parameter. This approach also has the downside of requiring a predefined selection of discrete parameters to choose from. The final strategy tries to avoid some of these restrictions, namely the *p* factor increase and the need to use discrete parameters.

6.1 Canary

In this strategy, we assign a certain number of the smallest buckets to be "canary buckets". When querying, we start by querying the canary buckets of all the data structures/parameters. These buckets will give an indication of the performance of the different parameters. The parameters can then be compared on a query-by-query

p_1	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9
p_2	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9
p_3	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9

Figure 6.1: The canary strategy, red buckets are queried. Using three canary buckets, and selecting p_2 based on the results.

basis, and the one with the lowest cost is selected. The rest of the buckets in the selected data structure can then be queried. This has the appealing property, on paper, that it should handle different query sizes as well. One could look at this strategy as a sort of semi-dynamic solution, trying to find the best static parameter by testing a small subset for each parameter before making a choice. A downside is therefore that it can not perform better than the best static choice.

As this is a query-time strategy, it introduces some additional complexity to each query by the fact that some buckets will effectively be queried multiple times. This is the explicit cost of the canary strategy and must be included in the strategy cost. The canary buckets should however only make up a small portion of the total number of elements, and the additional cost should therefore be low. It is important to keep in mind that even though this is a query-time strategy, there is still the p factor increase to insertions.

Using p different parameters and with n number of buckets, where $m < n$ buckets are canary, the maximum number of elements a structure can contain is

$$|\mathcal{B}_n| \leq 2^{n+1} - 1$$

Given p different data structures with m canary buckets, the maximum number of *additional* elements that may need to be searched is

$$|\mathcal{C}_m| \leq (p-1)\sum_{i=0}^{m} 2^i = (p-1)(s^{m+1} - 1)$$

To get an estimate of how many more (relative) elements may need to be searched when using the canary strategy with m canary buckets:

$$\begin{aligned} \frac{|\mathcal{C}_m|}{|\mathcal{B}_n|} &\approx \frac{(p-1)(s^{m+1}-1)}{2^{n+1}-1} \\ &\approx (p-1)\frac{2^{m+1}}{2^{n+1}} = (p-1)2^{m-n} \end{aligned}$$

This is an approximation of the relative increase in the total number of elements that can be searched when using the canary strategy. This indicates that as long as m is

p_1	**B_1**	B_2	**B_3**	B_4	B_5	B_6	B_7	B_8	B_9
p_2	B_1	**B_2**	B_3	**B_4**	**B_5**	**B_6**	**B_7**	B_8	B_9
p_3	B_1	B_2	B_3	B_4	B_5	B_6	B_7	**B_8**	**B_9**

Figure 6.2: The selection strategy, red buckets are selected and queried.

sufficiently smaller than n, the cost will be manageable. It also provides a bound for how large the performance gap between different potential parameters for a specific data structure should be, for the data structure to be worth considering. As an example, consider the case with $p = 3$, $n = 17$ and $m = 8$, the explicit cost can be in the magnitude of $(p-1)2^{m-n} = 0.00390625 \approx 0.4\%$. This is however just a quick approximation, and the result from the queries will in reality probably differ. This is because a query is at worst an $\mathcal{O}(n)$ operation, but often sub-linear. This implies that a larger portion of smaller buckets (the canary buckets) will be used for distance calculations in comparison to larger buckets, which means that the approximation works better for near worst-case scenarios. This is in addition to the fact that the estimation doesn't take into account that the different parameters will perform differently.

Challenge 1 can be tested by varying the number of canary buckets and finding an appropriate trade-off between accuracy and overhead. The issue of challenge 2 will need to be addressed, as this is a clear example of a strategy that relies on the transfer of optimal parameters to successors. This can be done with one of the mitigation methods. This strategy avoids challenges 3 by testing every parameter for each query. Challenge 4 is handled by querying the canary buckets with the user-supplied query. The challenge 5 is, like 2, an apparent problem for this strategy. Implementing the proposed retention policy would help with this.

Based on hypothesis 2, the expectation is that this strategy will perform better on uniform datasets than local ones.

Hypothesis 4 *Strategies that use smaller buckets to make guesses about larger ones will perform better at uniform datasets than local ones.*

6.2 Selection

As a counterpart to canary, this strategy looks at what would happen if the selection was made at insertion time. Each time a bucket is built, run k benchmark queries on it, and select the parameter with the best score to be used until the next time the bucket is built.

As opposed to the canary strategy, this strategy can select different parameters for different buckets, and is therefore able to (at least in theory) achieve better results than the best static choice. The strategy adds no explicit cost to queries, as all the overhead is associated with insertions. It is also noteworthy that it is possible to have cheaper "batch inserts", where the test queries only has to be done once, instead of for each new element. It is challenging to give an estimation of the impact that an insertion-time strategy has on insertions, as we try to optimize for efficient queries and efficient insertions does not relate to queries. Queries are however less expensive asymptotically, so the impact on insertions should be manageable.

It becomes obvious that even though the initial premise of the strategy is to keep track of multiple versions of the data structure, we actually only use one version of each bucket after it is selected. This means that we could try to ease the p-factor space cost by temporarily building the p versions of each, but only keeping the selected one. This would however not help, as when there is only one bucket present it will contain all elements, and it will have to be built p times.

The space cost can still be avoided if we built p versions of the bucket using the same space (instead of p duplicates). This includes "destroying" each data structure after it is built to reuse the space for the next one, and then recreating the selected one. This results in a trade-off between space cost, and the additional complexity of having to rebuild the bucket $p+1$ times instead of p. Note that if the last benchmarked bucket is the one that ends up being selected, we don't need to destroy it just to build the same one again, and we don't have the additional cost. One could imagine an approach where information from smaller buckets or previous rebuilds is used to make informed decisions about which parameter to be tested last.

If the overhead added to insertions become an issue and we want to achieve a lower insertion and space cost, we can use the same approach as used in canary. This can be done by having the largest bucket selected the most used parameter from smaller buckets, instead of searching for a new one itself. This would however introduce challenge 2 to the strategy that initially only looks at each bucket isolated. Another way of reducing insertion cost is by accumulating the results about the different parameters over time, and therefore not needing to run as many queries each time.

This strategy approaches the challenges in a notably different way than canary. Specifically, the challenges about translating parameter to successors (challenge 2) and the data structure being top-heavy (challenge 5) are not issues here, as all experiments and observations are local to the specific bucket. Challenge 3 is, in the same way as with canary, not a problem as each parameter is tested each time a bucket is rebuilt. The parameter k can be used to find a trade-off for challenge 1, how the data is collected (challlenge 4) as discussed earlier.

6.3 Rolling-average round-robin (RARR)

Both the previous strategies add an explicit cost to either queries or insertions. Here we explore a strategy that tries to do neither[1]. For each bucket size, keep track of a rolling average of the observed query cost and a selected parameter. When querying, use the selected parameter for each bucket. If the currently selected parameter performs worse than the rolling average, select the next one to be used for the next query. When a new bucket is added, a random parameter is assigned as a starting point.

There is some variance in query cost for different queries on the same parameter, so a bucket will not be stuck at the same parameter for a long time, even the optimal one. It will, however (hopefully) spend more time at the optimal parameter than the others. In that way, this strategy differs from the earlier described methods as well, as this strategy only tries to have a low cost on average and not for each query. Algorithm 1 describes the query operation for the RARR strategy. Here the variable $\beta \in [0, 1]$ decides how much influence newer observations have on the rolling average.

It can however be problematic that the largest bucket spends *any* time on bad parameters, so buckets larger than a *cutoff* could follow the best-found parameter so far. This can be viewed as a version of canary where only one parameter is tested for each bucket at each query, and therefore the extra cost from the canary buckets is avoided. But the canary strategy gathers more information about each parameter, so the RARR should find another way to gather more information before making a decision. The solution that ended up being implemented was introducing a *streak* counter that increased each time a query performed better than the rolling average for that bucket, and was reset each time a new bucket was selected, Then each parameter is weighted with the streaks, and the best is selected for buckets above the cutoff.

This shares the property from canary that there are no assumptions about query sizes. Even though it doesn't optimize for each individual query, it would adapt to trends in queries.

The RARR, with cutoff implemented, is more similar to the canary in terms of approach to challenges. If cutoff was not implemented, challenge 2 wouldn't be an issue. But the cutoff is needed to be able to find a suitable trade-off for challenge 1. The RARR should handle top-heavy states (challenge 5) better than the canary though, as the scores are accumulated over time, so previous observations can be used if there are no smaller buckets available.

[1]Other than the p factor insertion increase.

Algorithm 1 Query without implementing cutoff

```
function QUERY($q$)
    $result \leftarrow []$
    for $i \in [0,..,\mathcal{B}.numBuckets]$ do
        $selected, score, streak \leftarrow rollingAverages[i]$
        if ISFULL($\mathcal{B}[i][selected]$) then
            $elements, cost \leftarrow$ QUERYBUCKET($\mathcal{B}[i][selected]$, $q$)
            APPEND($result$, $elements$)
            if $score = -1$ then                                ▷ Scores are initialized to $-1$
                $score \leftarrow cost$
            else if $cost \leq score$ then
                $streak \leftarrow streak + 1$
            else
                $streak \leftarrow 0$
                $selected \leftarrow selected + 1 \mod \mathcal{B}[i].length$      ▷ Round-robin
            end if
            $score \leftarrow score * (1 - \beta) + cost * \beta$            ▷ Rolling average
            $rollingAverages[i] \leftarrow (selected, score, streak)$
        end if
    end for
    return $result$
end function
```

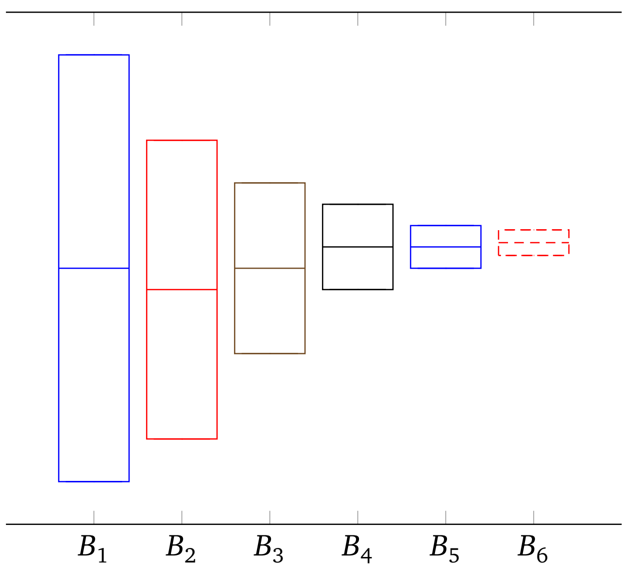

Figure 6.3: p_E and r changing as bucket size increase.

6.4 "Simulated" simulated annealing (SSA)

This strategy moves away from the idea of basing the selection on a set of predefined parameters but allows for the search of continuous parameters. The strategy is loosely based on simulated annealing, a probabilistic method for finding the global optimum for a function, where a *temperature* variable that decreases over time allows for the selection of possibly worse states in an attempt to not get stuck at a local maximum. In this case, however, instead of letting time control the temperature, the bucket size will.

Each bucket has an associated *range* consisting of an expected optimal parameter p_E, and a maximum distance r. The range gets smaller as the bucket size increases. p_E is derived from smaller buckets, and a bias related to that bucket. When a bucket is built, it selects a parameter at random in the range $[p_E - r, p_E + r]$. If the selected parameter performs "well", the bias will move towards that value, affecting the future p_E, as well as larger buckets in the future.

The SSA assumes that the cost is somewhat smooth and well-behaved when viewed as a function of the dataset. This is a trade-off between having discrete choices for parameters and allowing for search in a continuous range.

To decide if a parameter performs well, the same method as used in the selection strategy is used, namely running k test queries on the bucket after it is built. In this regard, the SSA strategy can be viewed as a version of selection, where we allow for continuous parameters and try to avoid having to build multiple versions of the same bucket each time. There is still an additional insertion cost, but this is only from doing queries, which as mentioned when discussing the selection strategy is less expensive asymptotically than the insertion. The SSA does however not have the constant factor p increase on insertions.

While the other strategies were somewhat straightforward to implement, this one required a bit more logic to get working. Each bucket benchmarks b parameters (requiring b rebuilds) before adding a bias to p_E. The first two parameters are selected in the range $[p_E - r, p_E]$ and $[p_E, p_E + r]$ to make sure that both sides of p_E are tested. After b parameters have been tested, the bias is moved towards the best-performing observed parameter, and the process is repeated. Algorithm 2 is used to combine the current range of B_i with the ranges of the smaller buckets, to create a new range for B_i. In this algorithm, the parameter $\beta \in [0,1]$ decides how fast the maximum distance r should shrink, and the parameter $\gamma \in [0,1]$ decides how much influence smaller buckets have on the new p_E.

Algorithm 2 Calculating the next range for a bucket after b tests

```
 1: function IterateBucketRange($P_E$, $R$, i)
 2:     if $i = 0$ then
 3:         return
 4:     end if
 5:     $p_E \leftarrow P_E[0]$
 6:     for $j \in [1,..,i]$ do
 7:         $p_E \leftarrow p_E * \gamma + P_E[j] * (1-\gamma)$
 8:     end for
 9:     $P_E[i] \leftarrow p_E$
10:     $R[i] \leftarrow R[i-1] * \beta$
11: end function
```

Algorithm 3 describes how algorithm 2 is used in combination with the bias from the benchmarks to iterate the range. The parameter $\sigma \in [0,1]$ decides how strong the effect of the bias should be, that is, how much closer to the best-tested parameter p_E should move.

The SSA approaches challenge 1 in the same way as selection, by allowing a variable number of test queries. It is however susceptible to challenge 2, but the hope is that the bias introduced by each bucket will help with that. If not, some of the mitigations can be applied. Challenge 3 is handled by the way random parameters within the range are tested. Challenge 4 is handled as with selection, and 5 the same way as RARR.

Algorithm 3 Adding a bias to p_E (for B_i) after b benchmarks

```
1: if $benchmarks[i].length = b$ then
2:     $p \leftarrow \min_{(p,score) \in benchmarks[i]} score$
3:     $P_E[i] \leftarrow P_E[i] + (p - P_E[i]) * \sigma$  ▷ Bias is introduced, moving $P_E[i]$ towards $p$
4:     $benchmarks[i] \leftarrow []$
5: end if
6: IterateBucketRange($P_E$, $R$, $i$)
7:
8: ...                                        ▷ Select new parameter and (re)build $B_i$
9:
10: $br \leftarrow$ RunBenchmark($B$, $i$)
11: Push($benchmarks[i]$, $br$)
```

Chapter 7

Experiments

7.1 Implementation

The algorithms tested here are implemented in Rust[16]. Rust is fast and has a flexible type system with good support for generic types, which makes it especially suitable for this task where the algorithm is the same for all underlying data types and data structures. Considering the definition of the MetricIndex trait shown in listing 7.1, the only thing needed to extend an index to a new data type is to implement a distance function for the given type.

Code listing 7.1: MetricIndex API

```
pub type DistFunc<I> = fn(&I, &I) -> f64;

pub struct Query<I> {
    pub center: I,
    pub r: f64,
}

pub trait MetricIndex: Sized {
    type Parameters;
    type Item;

    fn build(
        dataset: Vec<Self::Item>,
        dist_func: DistFunc<Self::Item>,
        params: &Self::Parameters,
    ) -> Self;
    // Returns the elements and the cost (number of distance calculations)
    fn query(&self, q: &Query<Self::Item>) -> (Vec<&Self::Item>, f64);
    // Combine multiple metric indexes (buckets) into a new one
    fn combine(parts: Vec<Self>, params: &Self::Parameters) -> Self;
}
```

In just the same way a generic MetricIndex can contain any data type as long as it provides a distance function, a Bentley-Saxe transformation can contain any metric index as long as it provides the functions described in the trait.

Code listing 7.2: Bentley-Saxe struct

```
pub struct BS<M, P, I>
where
    M: MetricIndex<Parameters = P, Item = I>,
{
    pub(crate) buckets: Vec<Option<M>>,
    dist: DistFunc<I>,
    params: P,
}
```

A generic insert and query function for the binary Bentley-Saxe transformation can then easily be implemented for any metric index, containing any type of elements as seen in listing 7.3.

Code listing 7.3: Binary Bentley-Saxe implementation

```
impl<M, P, I> BS<M, P, I>
where
    M: MetricIndex<Parameters = P, Item = I>,
    P: Clone,
    I: Clone,
{
    pub fn query(&mut self, q: &Query<I>) -> Vec<&I> {
        self.buckets
            .iter()
            .filter_map(|t| t.as_ref())
            .map(|t| t.query(q).0)
            .flatten()
            .collect()
    }
    pub fn insert(&mut self, p: I) {
        if self.buckets[self.buckets.len() - 1].is_some() {
            self.buckets.push(None);
        }
        let mut i = 0;
        let mut buckets = Vec::new();
        while let Some(bucket) = self.buckets[i].take() {
            buckets.push(bucket);
            i += 1;
        }
        buckets.push(MetricIndex::build(vec![p], self.dist, &self.params));
        let new: M = MetricIndex::combine(buckets, &self.params);
        self.buckets[i] = Some(new);
    }
}
```

The different strategies can be implemented in much the same way and allows for a large amount of flexibility when testing different combinations of data types and data structures. The code exemplifies how the implementations are independent of the underlying data structure, which can even allow for the use of 3rd party libraries where the algorithm can be treated as a black box.

The underlying HC and Bentley-Saxe implementations used were taken from the preliminary for this project, and the different strategies were implemented as "wrappers" in the same way as shown in listing 7.3. For the strategies that wrapped the Bentley-Saxe algorithm itself, some small modifications were necessary to for example be able to query a specific bucket for the canary and selection strategies.

7.2 Results

This is not meant to be a deep dive into the specifics of the different strategies, but rather as a way to explore the research questions and the BS-OPTPARAM using these strategies. Only the query costs will be investigated in this section. The insertion cost is, as mentioned earlier, harder to draw conclusions from. It is worth noting however that in all configurations tested in these experiments, the overhead to the insertion cost was manageable.

The datasets used for these experiments are either randomly generated, using the rust package rand[17] to provide utilities for random number generation, or taken from the International Conference on Similarity Search and Applications(SISAP)[18] to get some real-world datasets. The SISAP dataset used will be referred to as the NASA dataset and contains 40150 20-dimensional points. The randomly generated datasets are:

- 8-dimensional random points within the unit hypercube
- 8-dimensional trend with local picking method
- 8-dimensional trend with uniform picking method

The *trend* is created by creating points randomly around a center that moves as more points are created. This creates a dataset that looks like 5.1. The *trend strength* decides how "stretched" out the dataset should be. The local picking method inserts the elements in the same order as they were created, while the uniform picking method shuffles them first.

The underlying data structure for all tests is the HC using the Euclidean distance function. The parameters $\alpha \in [1.0, 1.5, 2.0]$ are provided as a choice to the strategies and will be used when calculating the C_{Average}. The random datasets contain $2^{17}-1 =$

R_{Average}	−10%
R_{Best}	8%
Average cost	43645
Canary overhead	451
Correct	69%

Table 7.1: Canary with random uniform data set, using 8 canary buckets

131071 elements (for 17 buckets, all full) unless otherwise stated. The queries used to test the different strategies use a random element from the dataset as query center and retrieve about 1% of the total dataset. Each data point is the average of the cost from 1000-10000 queries. The cost is calculated by counting the number of times the distance function is called when running a query.

In the graphs presenting strategy costs, a lower value is better because of the way strategy costs are defined.

7.2.1 Canary

If when executing a query, the canary buckets obtain the costs $[219, 223, 193]$ for the parameters $[1.0, 1.5, 2.0]$ respectively. The algorithm will then select parameter 2.0 as the parameter with the lowest cost. The total cost for running all these canary queries is $219 + 223 + 193 = 635$, but as one of the parameters has to be selected anyways, we can describe the *canary overhead* as this sum minus that of the selected parameter; $635 - 193 = 442$. For all buckets the costs were $[41325, 29836, 25590]$ respectively, and the algorithm was able to pick the best choice of parameter. But to be able to aggregate and compare the results from multiple queries, The strategy cost is used.

$$C_{\text{Average}} = \text{cost of selected} + \text{canary overhead} - \text{average cost}$$

$$C_{\text{Best}} = \text{cost of selected} + \text{canary overhead} - \text{best cost}$$

This expression clearly states what has been discussed earlier, that the potential gain from selecting the correct parameter must be able to outweigh the overhead associated with a strategy. In this case, the strategy cost versus average was $C_{\text{Average}} = 25590 + 442 - 32250 = -6218$, which means that the strategy gave a better result than what is expected from a random selection.

Using this definition of the strategy cost, it becomes easy to aggregate multiple queries, and a negative average strategy cost will still indicate a better result than random selection. Using the same configuration, an average of 10000 queries gives the result shown in table 7.1.

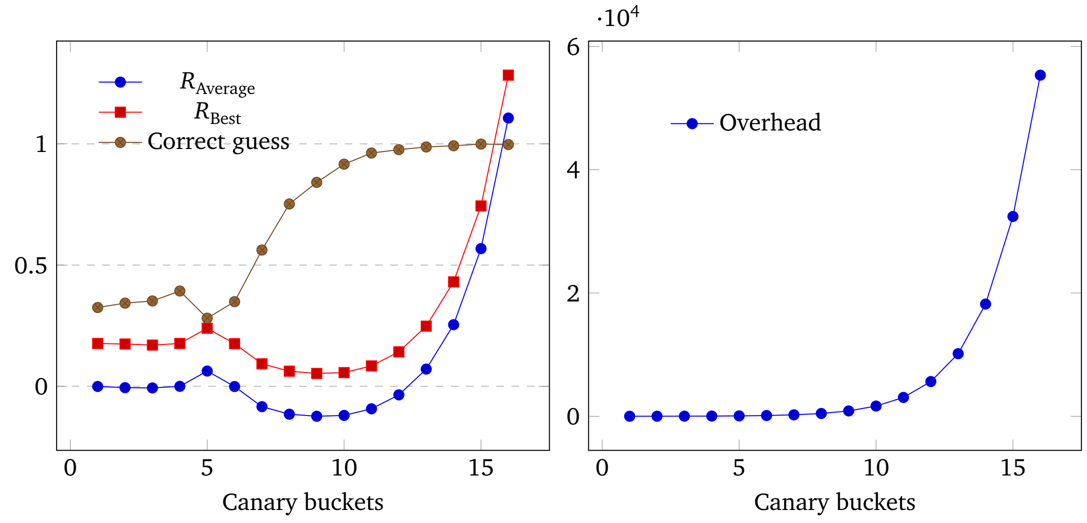

Figure 7.1: Canary, cost, fraction of correct guesses and overhead as number of canary buckets grow

This result shows a reduction in distance calculations of about 10% from a randomly selected parameter, but this is a very specific scenario with a somewhat large number of elements and where all buckets are full, which is a rare case. It does however indicate a promise for the retention approach to top-heavy structures, as the canary overhead seems to be a manageable part of the total cost. The cost compared to the best static choice is however quite bad, at about 8%. This is however somewhat expected, as the canary strategy will always be more expensive than the best static choice.

To explore how many canary buckets are needed, I ran the same test multiple times, using a varying number of canary buckets as shown in figure 7.1. The figure seems to indicate that when $|\mathcal{B}| = 17$ the optimal choice of the number of canary buckets is in the range of $8-10$. Note that the reason why there are correct guesses for such a low number of buckets is that when canary buckets give the same cost for different parameters, a random one is selected. Because the first buckets are so small, the costs often end up being the same, and we can expect to guess correctly with a $\frac{1}{3}$ chance. We also quickly see diminishing returns as new buckets are added.

It is however important to know how this optimal number of canary buckets changes as new elements are inserted, and new buckets are added. Figure 7.2 shows how the cost changes as new elements are inserted, and more buckets are added, using $\frac{n}{2}$ buckets as canary, where n is the total number of buckets. These benchmarks use random 8-dimensional points within the unit hypercube and are also tested at the points where all the buckets are full. The graph shows that negative (average) cost is achieved at 14 buckets. Note that each new bucket represents exponentially more

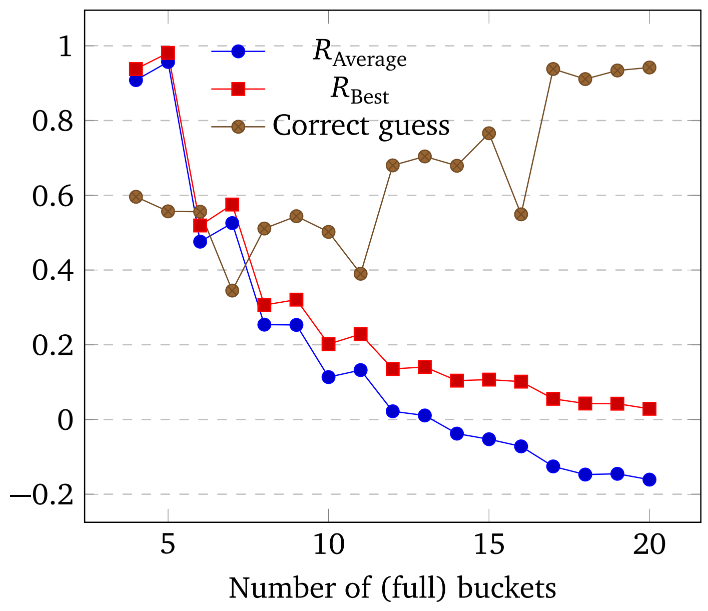

Figure 7.2: Canary, cost as size of data structure increases, using $\frac{|\mathcal{B}|}{2}$ canary buckets

elements, so while 14 full buckets represent $2^{14} - 1 = 16383$ elements, 15 full buckets represent $2^{15} - 1 = 32767$ and so on. Figure 7.3 shows the same benchmark, but where there are 8 canary buckets if $n \geq 10$, and none otherwise. Both these results indicate that as long as a certain number of elements are added, about 14 buckets worth, using about 8 canary buckets seems to give good results.

Testing on the NASA dataset, the canary strategy was still able to achieve a 10% reduction in distance calculations as shown in table 7.2. The NASA dataset contains 40150 elements as mentioned earlier. The binary representation of 40150 is

$$40150 = 0b1001110011010110$$

which means that there is a total of 16 buckets, but not all of them are full. The 8 canary buckets $0b11010110$ (the last 8 digits of the binary string) luckily contain enough elements to achieve good guesses, apparent from the results, but if the dataset contained $0b1000000000000000 = 32768$ elements, there would be no available canary buckets, and some mitigation method would have been needed.

As discussed earlier, the expected outcome when dealing with non-uniform picking methods is that a local picking method will perform worse than a uniform picking method over the same set of elements. Figure 7.4 shows the cost and correct guesses as the trend datasets are "dragged" out along a line, being uniform at 0. As expected, the fraction of correct guesses for the local picking method decreases as this effect increases. Surprisingly though, the uniform picking method performs even worse than the local picking method, exactly opposite what was expected. It is also interesting to

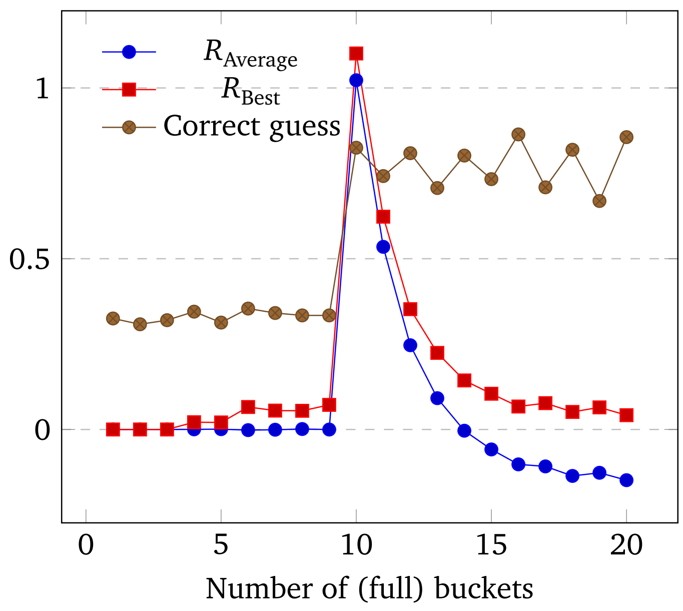

Figure 7.3: Canary, cost as size of data structure increases, using 8 canary buckets if $|\mathcal{B}| \geq 10$

R_{Average}	−10%
R_{Best}	4%
Average cost:	7937
Canary overhead:	42
Correct:	72%

Table 7.2: Canary with NASA data set, using 8 canary buckets.

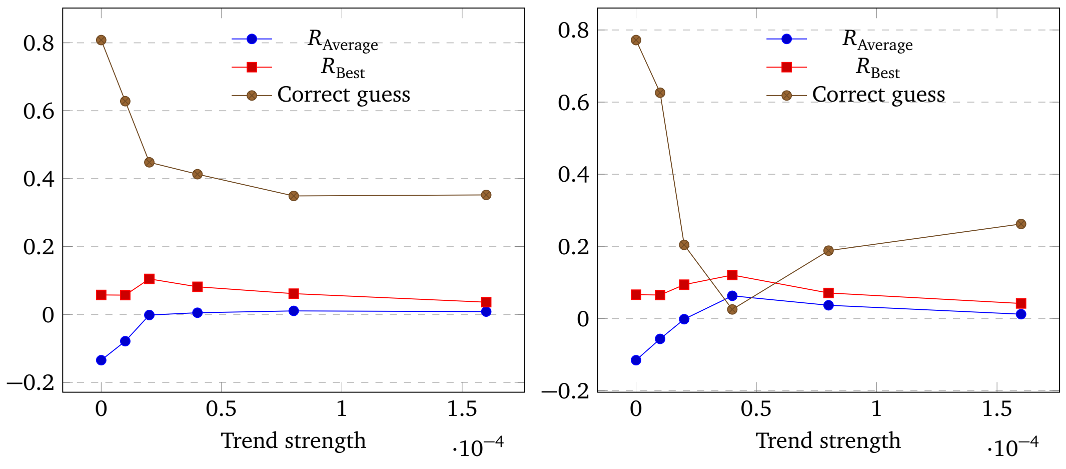

Figure 7.4: Canary, cost and correct guesses using local picking method (left) and uniform picking method (right) as trend in dataset is more exaggerated

R_{Average}	−18%
R_{Best}	0%
Average cost	43845

Table 7.3: Selection with random uniform data set

look at how the canary strategy actively chooses wrong parameters for the uniform picking method and performs worse than average, but with the local picking method, the worst results are about the same as the average.

This result indicates that there are some datasets that the canary strategy has problems with, but what characterizes these data sets is not known.

7.2.2 Selection

Using the same parameters and data set used in the initial canary test, and with $k = 100$ test queries, the strategy obtained the results shown in table 7.3. These results are notably better than the canary for the same dataset, showing that it finds and selects the optimal parameter, and achieving $R_{\text{Best}} = 0$. Note that the select strategy can, in theory, be able to outperform the optimal static choice. This result is however in line with earlier hypotheses that for a random uniform data set, the optimal selection will always be the best static choice. It also makes sense that this strategy performs well, as the way the test queries work and the way this benchmark is run are essentially the same.

Table 7.4 shows the select strategy used on the NASA data set. This strategy performed about the same as the canary for this data set, but notably worse than the

R_{Average}	-11%
R_{Best}	4%
Average cost	8075

Table 7.4: Selection with NASA data set

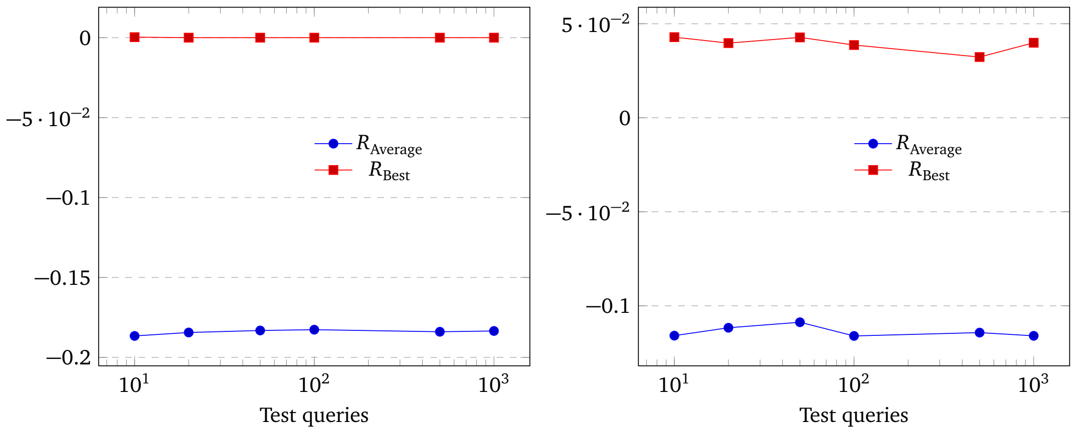

Figure 7.5: Selection, cost as number of test queries are increased, using random data set (left) and nasa (right)

random data set.

To this point, $k = 100$ has been used, as some small-scale testing indicated that it provided reliable results. Figure 7.5 shows cost as k increases, and it looks like there isn't a huge difference in the results. This indicates that it is possible to keep the insertion costs relatively low. This table also shows what was observed from table 7.3 and 7.4, where the select strategy agrees on the optimal static choice for the random dataset but makes some bad decisions when it comes to the NASA dataset.

Figure 7.6 shows the selection strategy tested against the same trend dataset that the canary used. It is interesting to see how the local picking method performs better when the trend strength is around 0.2 and 0.4, while the uniform picking method performed better at the highest strength. The difference in R_{Average} around 0.2 and 0.4 between the local and uniform picking methods may just come as a result of the difference in the performance of the parameters used getting smaller, and therefore the average and best gets closer. This theory is backed up by the fact that R_{Best} decreases while R_{Average} increases.

Although the select strategy performed better on the high-strength trend with the uniform picking method, the result does not have implications for the challenge 2 in the same way that was observed with the canary. This is because, as discussed earlier, the selection strategy doesn't relate small buckets to its successors.

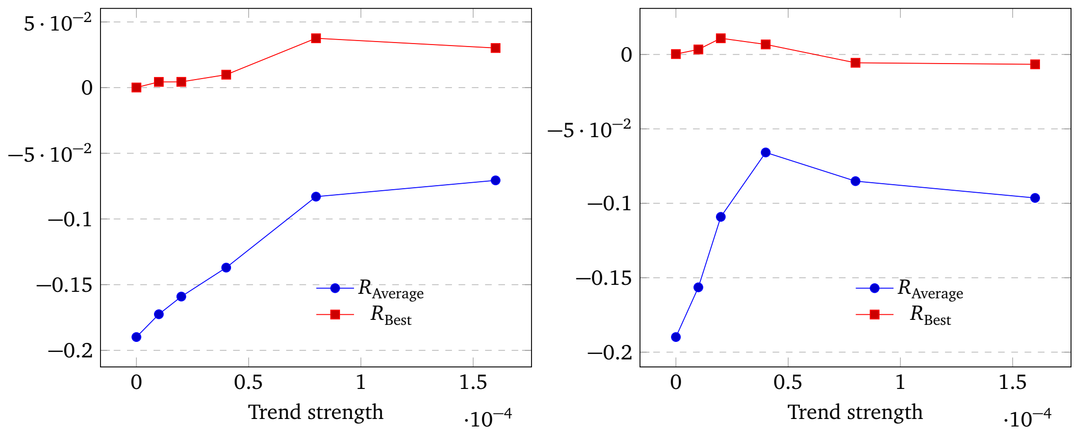

Figure 7.6: Selection, cost using local picking method (left) and uniform picking method (right) as trend in dataset is more exaggerated

R_{Average}	62%
R_{Best}	101%
Average cost	42401

Table 7.5: RARR without cutoff and $\beta = 0.2$

The uniform picking method in figure 7.6 shows a case where the select strategy actually outperforms the best static choice. This is an interesting result, a case where selecting different parameters for different buckets results in better performance than just sticking to the best parameter for all buckets. Although the difference is small and the scenario wasn't as easy to produce as initially expected, it is interesting to observe.

7.2.3 RARR

Table 7.5 shows results from the RARR strategy averaged over 10000 queries, using $\beta = 0.2$ and a cutoff of zero. As predicted, these results are horrible, with an increase of over 100% in distance calculations compared to the best static choice. This is the first result where the difference in potential parameters really shows. The only significant overhead this strategy introduces to queries is the implicit overhead of choosing the wrong parameter for a bucket. Given that with zero cutoff even the largest buckets regularly explore worse parameters, it is not surprising that the performance takes a hit.

To investigate this further and look into how low the cost can become, the strategy costs are plotted against different cutoffs in figure 7.7. This is in some ways analog-

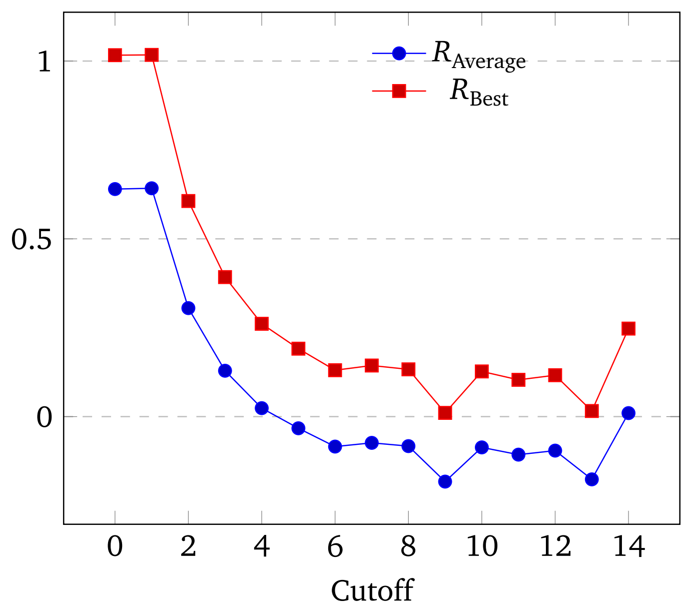

Figure 7.7: RARR with variable cutoff and $\beta = 0.2$

ous to the results from the canary strategy shown in figure 7.1, only with the x-axis flipped. We can see that the optimal cutoff when using 17 buckets seems to be 9, but one thing that should be pointed out here is that there was a lot of variance in the results when using the RARR strategy compared to the previous two, so the number of queries used for the averaged result had to be increased. In reality, I suspect that any cutoff in the range 7-12 would suffice.

Figure 7.8 shows cost using different values for β, the parameter that controls the weight of the rolling average. Surprisingly the NASA dataset gives more stable results than the random one and gave better performance compared to the best static parameter.

Figure 7.9 plots the trend datasets the same way as with earlier strategies. The same phenomenon is observed here as with selection, where the uniform picking method seems to perform better on weaker trends, while the local picking method performs better on stronger trends. Here we also see cases where the gap between the best and average shrinks, as discussed earlier. In this case, however, RARR is actually affected by challenge 2, but the results may be a bit too chaotic to be able to get much insight.

7.2.4 SSA

The SSA is in many ways the most complex of the strategies. This is reflected in that there are a lot of small dials to tune. For these experiments, the parameters were

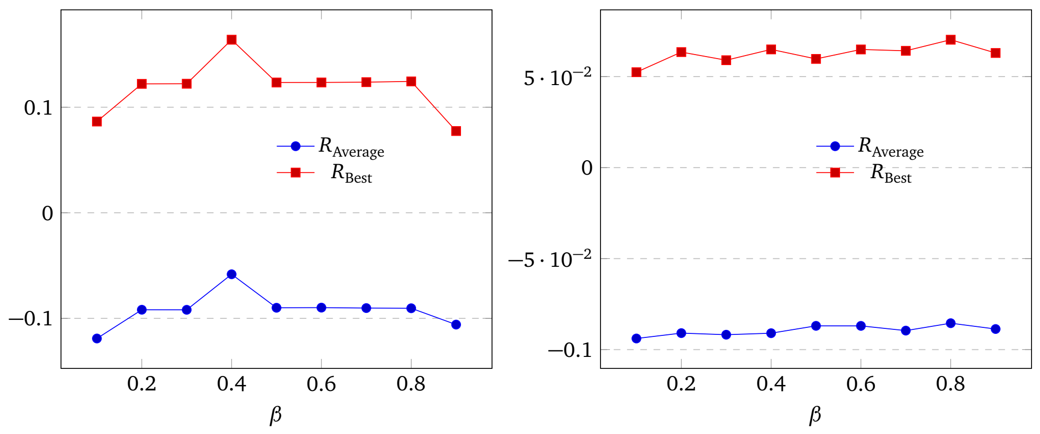

Figure 7.8: RARR with variable β with 9 as cutoff, with random dataset (left) and nasa (right)

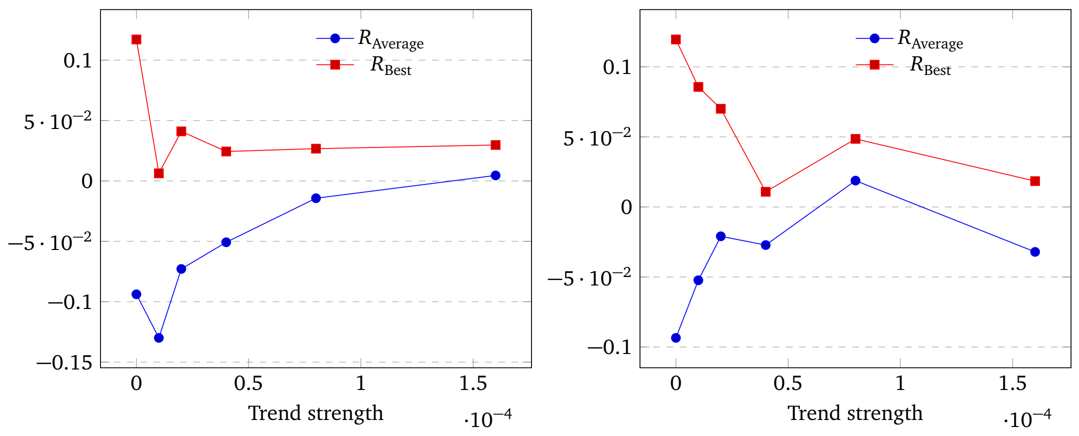

Figure 7.9: RARR, cost using local picking method (left) and uniform picking method (right) as trend in dataset is more exaggerated

R_{Average}	−20%
R_{Best}	−2%
Average cost	43878

Table 7.6: SSA with random uniform dataset

R_{Average}	−12%
R_{Best}	3%
Average cost	8128

Table 7.7: SSA with nasa dataset

selected by trial and error, as a deep-dive into the performance impact of the SSA related parameters would be too much. Using $\beta = 0.8$, $\gamma = 0.5$ and $\sigma = 0.5$ seemed to give good results for most datasets that were tested.

When evaluating the SSA strategy, it is compared against the same static parameters used in the earlier tests. This however means that compared to the other discrete strategies, the SSA can perform better than the best static choice by way of finding a better parameter. This can be seen in table 7.6, where the SSA outperforms the best static choice, even with a uniform dataset.

Comparing the results from the random dataset to the results from the NASA dataset shown in table 7.7, we see that the SSA performed significantly worse on the NASA dataset, but still at the same level, or even better, compared to the previously tested strategies. It is interesting to see how the performance has dropped though, and one possibility as to why may be that the NASA dataset has significantly fewer elements than the random one, and the SSA may need more insertions to have time to explore and adapt to the inserted data.

To test this theory the cost was plotted against the number of buckets, in figure 7.10, as new elements were inserted. Here we see that it takes about 13 buckets ($2^13-1 = 8191$ elements) for the SSA to find a parameter that performs better than the best static choice, so it seems like the result from the NASA dataset should have had enough insertions, and maybe the SSA just don't perform so well on that specific dataset. This is however with a random dataset, and for others, it may take more. So even though the NASA dataset has more elements, there is a possibility that it could have worked if the dataset was larger.

Figure 7.11 shows the usual test with the trend dataset. These results look similar to what has been seen earlier. Although if we compare the graphs for both picking methods, there may be a case for that they look more similar compared to each other than the other strategies do. This may be an indication that the SSA is more predictable than the other strategies.

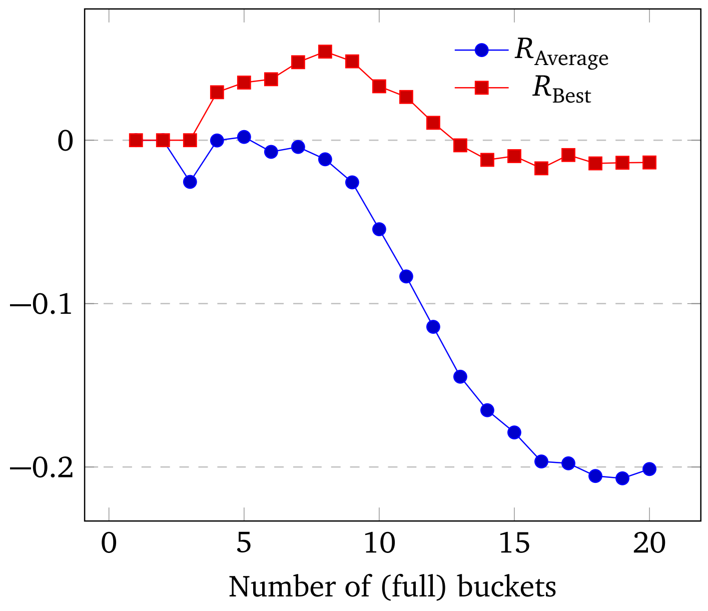

Figure 7.10: SSA with variable number of buckets/elements

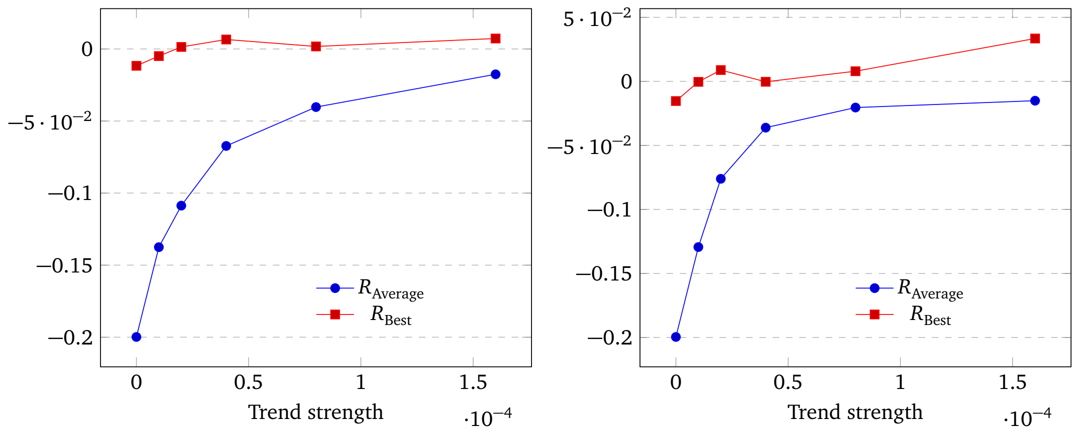

Figure 7.11: SSA, cost using local picking method (left) and uniform picking method (right) as trend in dataset is more exaggerated

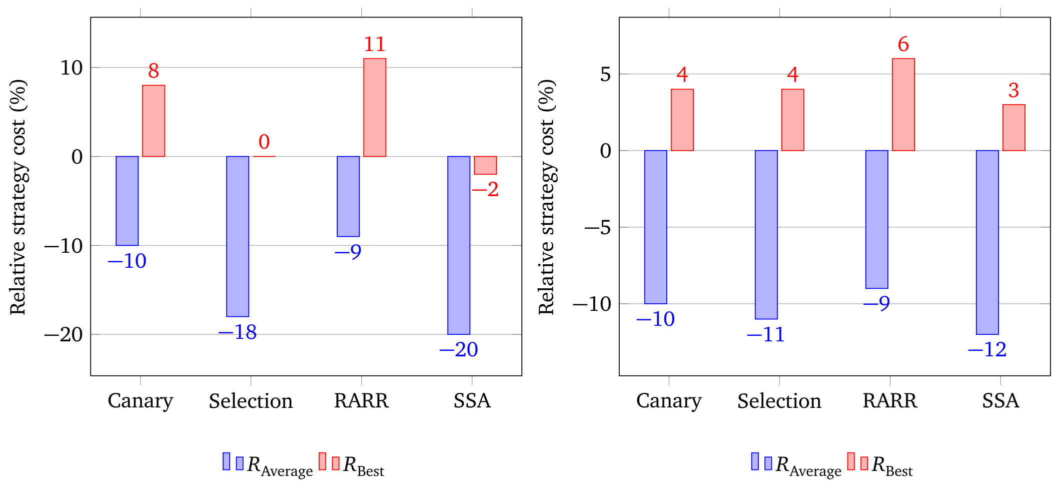

Figure 7.12: The results from all strategies compared, with random dataset (left) and nasa (right).

7.2.5 Summary

Figure 7.12 compares the observed strategy costs when using the random and NASA datasets for all strategies. According to the results, the insertion-time strategies (Selection and SSA) outperform the query-time strategies (Canary and RARR) on both datasets, with SSA performing the best and RARR performing the worst.

Chapter 8

Discussion

Looking back at the research questions, I would say that both questions were answered. I interpret the results here as Bentley-Saxe transformation successfully being applied to do this kind of optimization, and that the optimization was actually enough to achieve a significant reduction in distance calculations.

The amount of different combinations of strategies, possible datasets, parameters, mitigations from challenges, and parameters for the strategies that could be examined here is staggering, which made it difficult to do exhaustive tests and analysis of the different strategies. They did however work well as a method of testing out the general BS-OPTPARAM approach.

8.1 The strategies

While the results were mixed, and not thorough enough to draw any hard conclusions from in regards to the different strategies, they shone a light on multiple interesting areas that may be worth continuing investigating. Maybe most notably that hypothesis 4 did not seem to hold, or that it at least wasn't as simple as first assumed. This also made mitigations aimed at handling challenge 2 more difficult to test out, which is another reason why it wasn't included here.

It was useful to include the strategy cost compared to both the average and best static choice, as the R_{Best} adds context to the R_{Average}. If both weren't included, there would have been places where it looks like the average cost gets worse, but it is in reality just the performance of the different parameters getting closer to each other.

The selection strategy performed well and is in some ways the strategy with the simplest objective. I was however surprised at how the strategy failed to identify optimal buckets when using the NASA dataset. It is interesting to note how the trend dataset with a local picking method actually performs worse as the trend strength is increased, even though it doesn't relate buckets of different sizes to each other, so challenge 2 isn't a factor. What is interesting about this is that each bucket is trained with the data it contains itself, while the queries used in the experiment are taken from the entire dataset. As the picking method is local, the different buckets will contain a subset making up different, almost disjunct regions. This is opposed to the uniform picking method where each bucket spans approximately the same region. One would assume that because of this a lot of buckets can discard the queries early as they often are outside the region defined by the query.

It was hard to get much insight into how the RARR performed as the results varied a lot. This was probably because of the randomness introduced by the queries and rolling average dynamic. I expected that this would be preventable by using a small β, to slow down changes to the rolling average, but it apparently wasn't enough. Two things I would have tried in the next iteration of the RARR is to have a mechanic where the score rises quickly, but falls slowly, and where the selected parameter moves to the next randomly, with a chance that is based on how much worse than the rolling average the observed cost is.

The results for the SSA were a pleasant surprise. Even though it has the potential for a lot of tweaking, it seems like the SSA parameters found and used worked reasonably well for all the datasets it was tested with.

It was interesting to find a couple of cases where the BS-OPTPARAM was able to beat the best static choice. It can be discussed whether or not the results from the SSA should be included in this category, as the reason why the strategy performed better when using a random dataset was that it found a new parameter that performed better than the options it was compared against. These are also interesting results, but it is not the same as when the selection strategy performed better than the static best, where that result came from different buckets using different parameters.

8.2 The chosen parameters and the strategy cost

The assumption and premise for this project is that we have no knowledge about the data set, and if this is true any potential parameter may, by definition, be the "correct" one. By this logic, it makes sense to compare to the average of a selection of potential parameters and in this way, the R_{Average} can be used if we accept the premises. In most real-world cases, however, we can often assume that we have some insight into the dataset, so a semi-static baseline is more reasonable. This is a reason why comparing against the best static parameter in the set may be better suited for communication and comparison purposes.

A problem with the way these experiments have been done is that there has to be made a decision about which parameters should be used. This set of parameters can have an impact on the resulting strategy cost as including worse parameters can "punish" wrong choices made by the BS-OPTPARAM more than if all parameters were somewhat good. Even though all parameters used in the experiments were found to be optimal for some dataset, and therefore they are all potential parameters, one could make a different selection of potential parameters to achieve better or worse strategy costs, and a strategy cost of $R_{\text{Average}} = -5\%$ can be turned into a strategy cost of $R_{\text{Average}} = -10\%$ by simply choosing a different set of parameters to compare to. There is however a case to be made that as long as the BS-OPTPARAM strategy has to make a choice between the same parameters that it is compared against, the effect of different sets of parameters only exaggerate the result, but a positive result will still be positive.

8.3 Other things of note

When considering the different strategies compared to each other, there are some details that may not be evident from the results themselves. All strategies introduced some new parameter which, unless the parameter is easy to find a global optimum for, just moves the problem of finding the optimal α for the HC to finding the optimal parameters of the auto-tuning strategy. Although the canary strategy didn't exactly stand out, it is arguably the easiest to implement (about the same as selection) and requires the least amount of tuning. It is also important to note that the insertion-time strategies require knowledge about what query size is expected to be used, in order to be "trained" properly, while the query-time strategies didn't assume anything about the queries. This is however a property that results from the underlying HC data structure, and probably most metric indexes, and not the BS-OPTPARAM itself, so there may be applications where this is not the case.

As mentioned earlier, this project doesn't examine the complexity of insertions other than giving some estimations about increased cost, and actually running the experiments and seeing if there were any extreme outliers, which there were not. This is justified by the fact that the objective is to optimize for query cost, which is at the expense of build complexity even when just using the HC. When considering the HC, the build complexity goes from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n^2)$ when the imbalance increases. The explicit insertion-time cost of a strategy is given by the definition of the strategy, like the test queries in selection, but we could also consider an implicit insertion-time strategy as a parallel to the implicit query-time cost of selecting the wrong parameter. If a strategy builds buckets using every available parameter, like selection, and some of these parameters are unnecessarily imbalanced in regards to the data inserted, the complexity of the operation may end up being asymptotically worse.

The use of Rust for the implementation of these algorithms was both a positive experience, but with some predictable downsides. Some of the positives were mentioned earlier. The ownership and memory management models makes it easier to find possible bugs early, but it also requires some mental gymnastics to be able to implement some data structures.

8.4 Future work

There are many ways this work can be continued. All the proposed algorithms could probably benefit from continued work with deeper insight into why the different datasets behave as they do, as well as a reevaluation of the assumptions used for this project. I think personally that query-time strategies that adapt to both insertions and queries are the most interesting because the optimal choice for the HC is dependant on both. There may however be other data structures where this is not the case, and insertion-time strategies are more suited, so one way to continue the work would be to both look into new underlying data structures within the field of metric indexing, and look at new applications of the general BS-OPTPARAM procedure.

It would be interesting to get more insight into what characterizes the datasets that perform bad, and what new challenges they represent. Are new assumptions needed or can they be handled using better BS-OPTPARAM strategies designed with them in mind?

One thing that isn't looked at in this report but would be interesting is to explore is to model trends in the queries themselves, where we expect future queries to look somewhat like the ones that came before them, and therefore try to optimize for a more specific type of query instead of a random one. A simplified version of this would be to assume that the dataset is known, but how the dataset is going to be queried isn't, and use a query-time strategy to adapt a static data structure to different queries.

It would also be interesting to look into other uses of the Bentley-Saxe transformations in combination with different algorithm-specific optimizations, outside of finding optimal parameters. One thing that comes to mind is using the same idea of rebuilds and testing smaller buckets to do better pivot selection for the VP-tree or the HC, or possibly better cluster centers for the SSS-tree.

Chapter 9

Conclusion

This report defined the BS-OPTPARAM, a general approach to finding the optimal parameter for a metric indexing data structure using Bentley-Saxe transformations, as well as four different algorithms using this approach. The algorithms were implemented and tested using a couple of different datasets to get an idea of the general performance of the solutions, as well as some tests specific to some of the algorithms to explore their limitations.

The results show that the algorithms, with only a few exceptions, perform better than what would have been expected from a randomly selected parameter, and a few times even better than the best static choice for the parameter. This shows how the BS-OPTPARAM can be a useful tool when trying to solve this type of problem.

Acknowledgements

I would like to thank my supervisor Magnus Lie Hetland for providing great support and guidance throughout the year and this project.
I would like to thank Helene and my roommates for keeping me sane during the pandemic.
I would like to thank Webkom for providing a great environment to learn, and making my time at NTNU so great.

Bibliography

[1] K. Fredriksson, 'Engineering efficient metric indexes', *Pattern Recognition Letters*, vol. 28, no. 1, pp. 75–84, 2007.

[2] M. L. Hetland, 'Metrics and ambits and sprawls, oh my', *Lecture Notes in Computer Science*, pp. 126–139, 2020, ISSN: 1611-3349. DOI: 10.1007/978-3-030-60936-8_10. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-60936-8_10.

[3] P. N. Yianilos, 'Data structures and algorithms for nearest neighbor search in general metric spaces', in *Soda*, vol. 93, 1993, pp. 311–21.

[4] N. Brisaboa, O. Pedreira, D. Seco, R. Solar and R. Uribe, 'Clustering-based similarity search in metric spaces with sparse spatial centers', in *International Conference on Current Trends in Theory and Practice of Computer Science*, Springer, 2008, pp. 186–197.

[5] B. Naidan and M. L. Hetland, 'Static-to-dynamic transformation for metric indexing structures (extended version)', *Information Systems*, vol. 45, pp. 48–60, 2014.

[6] J. L. Bentley and J. B. Saxe, 'Decomposable searching problems i: Static-to-dynamic transformation', *J. algorithms*, vol. 1, no. 4, pp. 301–358, 1980.

[7] J. B. Saxe and J. L. Bentley, 'Transforming static data structures to dynamic structures', in *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, IEEE, 1979, pp. 148–168.

[8] O. Pedreira and N. R. Brisaboa, 'Spatial selection of sparse pivots for similarity search in metric spaces', in *International Conference on Current Trends in Theory and Practice of Computer Science*, Springer, 2007, pp. 434–445.

[9] M. Muja and D. G. Lowe, 'Scalable nearest neighbor algorithms for high dimensional data', *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[10] E. Chávez and G. Navarro, 'A compact space decomposition for effective metric indexing', *Pattern Recognition Letters*, vol. 26, no. 9, pp. 1363–1376, 2005.

[11] W. A. Burkhard and R. M. Keller, 'Some approaches to best-match file searching', *Communications of the ACM*, vol. 16, no. 4, pp. 230–236, 1973.

[12] W. Dong, Z. Wang, W. Josephson, M. Charikar and K. Li, 'Modeling lsh for performance tuning', in *Proceedings of the 17th ACM conference on Information and knowledge management*, 2008, pp. 669–678.

[13] E. Jääsaari, V. Hyvönen and T. Roos, 'Efficient autotuning of hyperparameters in approximate nearest neighbor search', in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, 2019, pp. 590–602.

[14] D. D. Sleator and R. E. Tarjan, 'Self-adjusting binary search trees', *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985.

[15] A. Andersson, 'General balanced trees', *Journal of Algorithms*, vol. 30, no. 1, pp. 1–18, 1999.

[16] (May 2021). 'Rust 1.51.0', [Online]. Available: `https://www.rust-lang.org/`.

[17] (May 2021). 'Rand 0.7', [Online]. Available: `https://crates.io/crates/rand`.

[18] K. Figueroa, G. Navarro and E. Chávez, *Metric spaces library*, Available at http://www.sisap.org/metricspaceslibrary.html, 2007.